Distributed search trees

Fault tolerance in an asynchronous environment

Originally published in:
Theory of Computing Systems 36(6), https://doi.org/10.1007/s00224-003-1121-7

Theory Comput. Systems **36**, 611–629 (2003)
DOI: 10.1007/s00224-003-1121-7

Theory of Computing Systems

Distributed Search Trees: Fault Tolerance in an Asynchronous Environment*

Konrad Schlude,[1] Eljas Soisalon-Soininen,[2] and Peter Widmayer[1]

[1]Institut für Theoretische Informatik, ETH Zentrum,
CH-8092 Zürich, Switzerland
{schlude,widmayer}@inf.ethz.ch

[2]Laboratory of Information Processing Science,
Helsinki University of Technology,
Otakaari 1 A, FIN-02150 Espoo, Finland
ess@cs.hut.fi

Abstract. We propose a distributed dictionary that allows insert and search operations and that tolerates arbitrary single server crashes. The distinctive feature of our model is that the crash of a server cannot be detected. This is in contrast to all other proposals of distributed fault-tolerant search structures presented thus far. It reflects the real situation in the internet more accurately, and is in general more suitable to complex overall conditions. This makes our solution fundamentally different from all previous ones, but also more complicated. We present in detail the algorithms for searching, insertion, and graceful recovery of crashed servers.

1. Introduction

The amount of digital information grows at a breathtaking pace, and constantly improving networking technology makes distributed computing power readily available. Modern databases make use of today's technology and develop into increasingly global information systems. The efficient storage and retrieval of data becomes a critical issue, and as a consequence, distributed data structures have gained considerable attention [1], [3], [6], [7], [9], [10], [12], [13], [15].

* A preliminary version of this report appeared as [14]. We gratefully acknowledge the support of the project "Highly available scalable data structures (2100-066768)" by the Swiss National Science Foundation SNF.

It has been shown to what extent classical central concepts carry over to the distributed setting: Litwin et al. [10] present a distributed hash file, whereas Kröll and Widmayer [6] propose a scalable data structure based on random binary leaf search trees. For many data warehousing and multidimensional database applications, the important access primitives include a variety of similarity search operations, such as nearest neighbor queries or the enumeration of the data in a close neighborhood of the query.

In their seminal paper [10], Litwin et al. coined the term *Scalable Distributed Data Structure* (*SDDS*) for structures that satisfy the following requirements: a file expands to new servers gracefully; there is no master site that must be accessed for virtually each operation; the file access and maintenance primitives never require atomic updates to multiple workstations.

1.1. *Fault Tolerance*

A distributed system offers the chance to remain operational even if an individual computer fails. For a large distributed system, such a failure must indeed be expected from time to time. Therefore, distributed data structures have been proposed that tolerate limited hardware failures and still support access to all data at all times. This failure resilience is achieved by means of data replication [9], or by applying the technique of parity records and buckets [8], [11]. In these proposals an essential feature is that a server can detect rapidly whether some other server is operational or crashed. This is certainly a reasonable assumption for local networks, but it is unrealistic for globally distributed databases.

The distinguishing feature of our approach is that fault tolerance can be achieved even if no server can ever detect whether some other server is operational or has crashed. This strong interpretation of asynchronicity makes the design of a data structure quite complicated and fundamentally different from the situation where a server crash can be detected [7]–[9], [11]. For instance, locks are not allowed to synchronize servers, since the server that has to unlock another server could have a crash failure, thereby letting the other wait—clearly an unacceptable situation.

1.2. *Results*

In this paper we propose a scalable, distributed dictionary. It is based on distributed search trees [6] that support *insert* and a variety of *search* operations for keys from a linearly ordered universe, say integers, in a challenging environment. The dictionary works in a totally asynchronous setting, where faulty servers cannot be detected, and it tolerates crashes: a single server crash is guaranteed to do no harm at all, and simultaneous crashes of more than one server are also often harmless, but with no guarantee. Such data structures are called *highly available* data structures [9]. More specifically:

1. The dictionary remains fully operational in the presence of a single failure, i.e., all search and insert operations work correctly and efficiently.
2. It enables efficient recovery, i.e., a server that has suffered a crash failure can reinvolve itself into the data structure, and after the recovery is complete, a client cannot distinguish whether a crash occurred.

3. Compared with the operations of the distributed search trees on which our approach is based, the high availability of our fault tolerant dictionary comes at a small constant overhead (number of messages, memory) in the worst case.

1.3. *The Model*

Let us now be more precise about the distributed system and its availability. Let the set of computers be connected by a network. Computers communicate by sending and receiving messages. Every computer is identified uniquely by its address. We do not focus on message size, assuming that large buffers for incoming messages are available. We express this by assuming that messages can be arbitrarily large, but we will not exploit this in any extreme way. In particular, fault tolerance will add nothing to the length of messages. Message transmission is asynchronous in the following sense. First, no computer has access to a global clock. Second, every sent message will be delivered eventually, but message transmission times are unpredictable and not bounded. Third, messages can pass each other on different paths. For instance, if a computer C sends a message m_1 and then a message m_2 to a computer C', then it is possible that C' receives m_2 before m_1.

We distinguish *client* computers that initiate insert or search operations on the data set from *servers* that store data. Starting with one server S_0, more and more servers become involved as needed to keep the individual server load small; we consider the set of potential servers to be arbitrarily large for this purpose. The SDDS conditions can be satisfied with the following basic strategy. An SDDS is distributed among servers and is manipulated by requests from client sites which always have their own image of the structure. However, the image of the client can be outdated, because the SDDS may have, for example, split some of its buckets and distributed them between old and new servers. The structure is designed so that with an outdated image the client can find the correct bucket but cannot send the query directly to the new server. After this, a new updated image will be sent to the client so that it cannot make the same error twice. In this way the most important property of an SDDS is achieved: no bottlenecks are created because clients with updated information can usually send their queries directly to correct servers.

In our model, communication is reliable, but a server can break down. Therefore, we distinguish *operational* and *crashed* servers. An operational server is fully functional, while a crashed server can neither receive nor send messages nor perform a computation. An operational server can crash. A crashed server loses all its data, except for some small piece of data in a secure memory. The reason is that, on one hand, a server cannot recover without any information at all about its role in the data structure, and, on the other hand, it is not economically feasible to protect all its data against loss. For simplicity of the discussion, we assume that the first crash of a server can only happen after the server has been involved into the data structure. This assumption is not a serious restriction, as we show in Section 5.1. Since a failure cannot be detected, we have to use a technique that can be classified as *Hot Standby* [5]; i.e., data is duplicated in a way where every key is stored on two different servers, and without failure both copies are equally well available. A crashed server can *recover*; this will change its *state* to operational, and the

server will reinvolve itself into the data structure in a way that we will propose in detail.

Server S_0 has an exceptional role and does not crash. This may appear like a strong and unrealistic assumption, but it is neither. It is realistic, because securing a single machine in a network is easy and is routinely done in practice; it is not stronger than the minimum needed, since the impossibility of distributed consensus [4] suggests that without this assumption, the desired behavior cannot be achieved. Furthermore, to obtain the required properties, we will see that our data structure also needs a secure and central entity, the so-called *split manager*. Although this is not desired in a distributed system, its existence is not a serious drawback. In fact, all distributed data structures in the literature need some central entity to do a small but important piece of work. For instance, in [6] such an entity finds the next server that can be involved into the data structure, and [10] needs a central entity as split coordinator. For convenience, we propose implementing the split manager within S_0, but other implementations are possible. We avoid the discussion of operational details that are local in a server, and assume that receiving and processing a message and sending messages as a consequence is one atomic step.

1.4. *Organization of This Paper*

Section 2 reviews the distributed binary search trees, and Section 3 presents our proposal for a fault-tolerant distributed dictionary based on distributed binary trees. Section 4 proves its properties. Section 5 discusses modifications, and Section 6 concludes the paper.

2. Distributed Binary Search Trees

We present a solution to the fault-tolerant distributed dictionary problem with distributed binary leaf search trees [6], in which the keys (and data) are stored in leaves that are similar to B-tree leaves and the internal nodes are binary routers. The tree structure not only supports the access of single records but also allows a variety of efficient similarity queries. We limit ourselves to the explicit discussion of the former, since it will become clear how to perform the latter. The nodes of the tree are stored in servers, and the edges are communication links. In this section we shortly review the binary tree structure, for more details we refer to [6].

With every node u, we associate a nonempty *responsibility interval* $I_u \subset \mathbb{Z}$, represented by a pair $l_u, r_u \in \mathbb{Z} \cup \{-\infty, \infty\}$. Node u is *responsible* for a key k, if $k \in I_u$. Each internal node u with left child v_1 and right child v_2 has a *split value* $\sigma_u \in I_u$ such that $I_{v_1} = [l_u, \sigma_u]$ and $I_{v_2} = [\sigma_u + 1, r_u]$. For every key $k \in \mathbb{Z}$ there is exactly one leaf v that is responsible for k. Leaves store keys, with leaf v storing a set $K_v \subseteq I_v$ of keys.

In our model, each server can hold a constant number of leaves. Internal nodes contain routing information only, and there is no a priori bound on the number of internal nodes a server can contain.

In a state in which no insert operations are under way, each client has a picture of that part of the binary tree structure above all leaves it has accessed. All searches

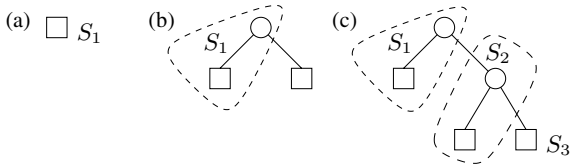

Fig. 1. The distributed tree grows from one leaf in server S_1 (a) to two leaves and servers (b), and further to three leaves and servers (c).

by clients will be directly sent to leaves and servers in the known part of the tree. If queries are evenly distributed, so is the load of servers; even if the distribution of queries and insertions is skewed, the load of servers can be leveled out nicely with a concept proposed in [15].

Initially, the structure contains one leaf only, and whenever, due to insertions, a split occurs a new server is taken to store one-half of the records the split node was responsible for. A split also means that a new internal node must be created; this will be stored in the old server that held the split node, see Figure 1.

To avoid the situation that the root becomes a bottleneck, the *lazy update concept* can be used. We illustrate this with the following example, more details can be found in [6]. Consider then the situation depicted in Figure 1, and assume that a client has an out-of-date picture of the structure telling that there is only one leaf even though the true picture is as given in Figure 1(c).

Assume further that the client searches for a key which has been moved to server S_3. Using its out-of-date picture the client sends the query to server S_1, and seeing that the required key is not in its own leaf, S_1 sends according to the tree structure the query to S_2. Similarly, S_2 sends the query to S_3 whose leaf is responsible for the key of the query.

After the query has been performed and the client has obtained the corresponding information, the search path used is also sent to the client. Then, for all keys the nodes in this path are responsible for, this path is first used in the client before accessing the global structure.

3. Highly Available Trees

Roughly speaking, the *Highly Available Tree* (*HAT*) that we now propose duplicates data and stores copies in two different distributed binary leaf search trees. Both trees are kept as similar as possible, but failures and different execution speeds may create differences between the trees that will be repaired as operations progress. Every node u in one tree has an associated node u' in the other tree, its *buddy*. Every request is performed on both trees simultaneously.

In more detail, a HAT H consists of a pair of rooted binary trees T_l and T_r with roots r_l, r_r. Initially, each of the trees $T_\bullet$ consists of its root $r_\bullet$ only. The trees grow by splitting a leaf into an internal node with two children.

We define the *buddy operator*, denoted by a prime symbol $'$, as follows. If r_l is the root of T_l and r_r is the root of T_r, then $r_l' := r_r$ and $r_r' := r_l$. If u is an internal node and

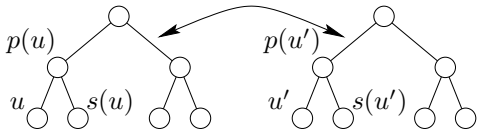

Fig. 2. Pair of two isomorphic trees.

v is its left (right) child, then v' is defined as the left (right) child of u'. For any other pair of nodes w_1, w_2, $w_1' \neq w_2$ and $w_2' \neq w_1$ hold. Observe that for any node u, we have $u'' = (u')' = u$. The parent of a node u is denoted by $p(u)$, the sibling of u is denoted by $s(u)$, see Figure 2. If $u' = v$ for two nodes u, v, then v is the *buddy* of u and u is the buddy of v.

It follows that a node has at most one buddy; a node can send a message to its buddy's address anytime, even though this buddy need not be part of the dictionary at the moment at which the message is sent. A node u knows the addresses of its sibling and its parent (if u is not a root). If u is an internal node, it knows the addresses of its children.

In H, two trees T_l, T_r are linked in the following way. Every node u is restricted to having the same responsibility interval as its buddy u', i.e., $I_u = I_{u'}$, and u knows the address of u'. Furthermore, u knows the address of its buddy's parent $p(u')$ and sibling $s(u')$. If u' is an internal node with children v_1' and v_2', then u knows the addresses of v_1' and v_2' or will know these addresses eventually.

A node u can send messages to node v, if u knows the address of v.

To simplify notation, we do not distinguish between an object and its address, i.e., u denotes a node or the address of this node, c denotes a client or the address of this client. If a node u receives a message m, then u is also informed about the address of the sender of m; in particular, u knows whether m was sent from its parent $p(u)$, its buddy u', or a client c.

3.1. *Mapping Nodes to Servers*

The described structure has to be mapped to the set of servers, i.e., every node u is stored on a server $S(u)$. The set of all potential servers is enumerated $\{S_0, S_1, S_2, \ldots\}$, but this enumeration and the corresponding addresses of the servers need to be known only to S_0. On the other hand, the addresses of S_0, S_1, and S_2 are known to every computer. Server S_0 maintains a counter new for the relative number of the server to be involved next. Since we start with two root servers, initially $new = 3$. A server S is called *involved* if there is a node u with $S = S(u)$. For the resilience against one-server failures it is important that a node and its buddy are not on the same server, i.e., $S(u) \neq S(u')$ for every node u. In order to ensure scalability, no server is allowed to store more than a constant number of leaves. Since internal nodes contain routing information only, we do not impose an a priori bound on the number of internal nodes on a server.

The allocation rule maps the nodes as follows (Figure 3):

1. The root of T_l is stored on S_1, the root of T_r is stored on S_2.

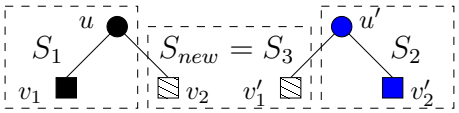

Fig. 3. Mapping trees to servers.

2. Let $u \in T_l$ be a leaf that performs a split and becomes an internal node with left child v_1 and right child v_2. Then:
   (a) The left child v_1 is stored on $S(u)$, i.e., $S(v_1) = S(u)$.
   (b) The right child v_2' is stored on $S(u')$, i.e., $S(v_2') = S(u')$.
   (c) The nodes v_2, v_1' are stored on a new server S_{new}, i.e., $S(v_2) = S(v_1') = S_{\text{new}}$.

The construction immediately implies:

Lemma 3.1. *The allocation rule of mapping HAT nodes to servers has the following properties*:

1. *Buddies are on different servers. More formally, for every node* u, $S(u) \neq S(u')$ *holds*.
2. *No server stores more than two leaves*.

Note that a message from a node u to a node v has to be sent from the server $S(u)$ to the server $S(v)$, and $S(v)$ has to hand the message over to v locally. We simplify notation by letting nodes act (instead of servers only) and therefore say that this message goes directly from u to v even if $S(u) = S(v)$.

It is possible that a node u is stored on a crashed server $S(u)$. To simplify notation, we call u *crashed*, if the server $S(u)$ is crashed or has not performed its recovery. Otherwise u is called *operational*.

3.2. *Dictionary Operations*

This section shows how the described structure can be used to implement the dictionary operations search and insert.

3.2.1. *Initialization.* A HAT H is initialized as empty. The empty structure H consists only of the roots r_l of T_l and r_r of T_r. Let r_l be stored on S_1, r_r on S_2. For the responsibility intervals, we get $I_{r_l} = I_{r_r} = \mathbb{Z}$. The two nodes do not contain keys, i.e., $K_{r_l} = K_{r_r} = \emptyset$. The address of r_l is stored in the secure memory of S_2, and the address of r_r is stored in the secure memory of S_1.

3.2.2. *Search.* A search request of a client c for a key k is performed simultaneously in both trees of H, since c cannot decide whether there is a failure in a tree. To perform an operation, c sends two messages to the roots r_l, r_r, from which the messages are forwarded to the responsible children, until the messages reach the responsible leaves. If a leaf receives a search request message, it sends its response about k to c, telling whether k has been inserted or not. Since in this naive approach the roots become bottlenecks,

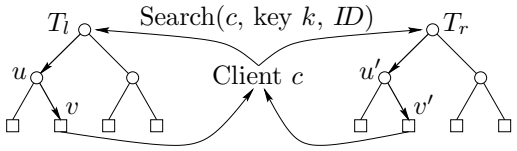

Fig. 4. Simultaneous search in a HAT.

we apply the ideas proposed in [6] of informing clients about the tree structure in a lazy fashion so that they can start future searches further down the tree. This is discussed in Section 5.2.

Whereas in an ordinary leaf search tree the search request leads to a search path from the root to the responsible leaf, in a HAT such a request defines a pair of search paths, see Figure 4.

However, as a consequence of the weakness of our assumptions about the distributed system, such a pair of independent search paths is not enough: if in Figure 4 the nodes $r_l = p(u)$ and u' crash, then the search request from client c cannot reach the responsible leaves. In order to bypass crashed nodes safely without loss of information, we connect the search paths in the following way. If a node u sends a message m to a child v, then u also sends m to v'. If there is no crash failure, v receives two copies of the same message, one from u, the other from u'. Theorem 4.5 proves that in this way, crashed nodes can be bypassed.

This, however, causes the next problem: if a node sends two messages for every message it receives, the number of messages doubles at every node on the search path. Since this exponential growth is totally unacceptable, we control it by keeping track of what happened as follows. If a client c wants to search for a key k, it chooses an identifier ID that is unique with respect to c, for instance the number of inserts and searches c has requested so far. Then c sends the message $Search(c, k, ID)$ to the roots of H. If a node receives a message, this node keeps track with a *message tag* for the pair c, ID of receiving this message, and it forwards the message to its corresponding child v and to v'. If a second copy of this message is received later, the node recognizes from the tag that it has forwarded the message before and hence does not forward the message again.

Now, a (smaller) problem is that a tag has to be stored in memory. To save memory, a tag should be deleted after a while. One option is to delete the tag after the node has received the second copy of the message (if the node is a root, then no tags are needed). However, now we run into a further problem: it is possible that a tag is never deleted. In the example of Figure 5, a search message is sent to buddies u and u'. Both u and u' are about to split. Now, assume that node u' has already performed its split, while u has

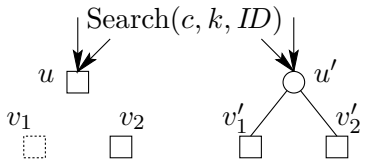

Fig. 5. Nonisomorphic trees.

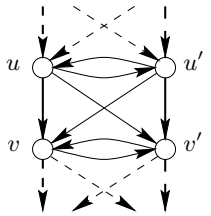

Fig. 6. Search ladder.

not. Since u is a leaf, it responds to the client. Moreover, since u' is an internal node, it forwards the search request to the responsible child v_2' and to v_2. The corresponding tags in u and u' will be deleted. However, the children v_2 and v_2' receive only one copy of the message, and, therefore, their tags will never be deleted. Although tags may seem like a rather small problem, their number might increase with every search operation, and, in the long run, this is clearly undesirable. We therefore choose to avoid the possible unbounded growth of the number of tags with the following concept.

If u forwards the search message to children v and v', or responds to a client c, then u sends the message together with the addresses of v and v' or c to its buddy u', called a *cross message*.[1] Lemma 4.1 shows that cross messages suffice to achieve the desired properties. We call the pair of connected search paths, including cross messages, a *search ladder*, see Figure 6.

In more detail, the tags work as follows. If a node u receives the c, ID pair in a message $Search(c, k, ID)$ for the first time, then u creates a *tag*. Such a tag consists of three entries:

- *Message*: The message identifier c, ID.
- *Received from*: The addresses of the nodes or the client c from which u has received the message in the past. Initially, this entry is empty. After having received the message $Search(c, k, ID)$ from a sender, u adds the address of this sender to the entry.
- *Sent to*: The addresses of the nodes or the client c to which the message is sent. If u is a leaf, then it sends messages to c and to u'. If u is an internal node, messages are sent to children v and v' and to u'.

If u is the responsible leaf for key k, then u sends a response message to client c. If u is an internal node, then u forwards the message $Search(c, k, ID)$ to its child v and its buddy's child v' with $k \in I_v = I_{v'}$. In both cases a message is sent to u', too.

If u receives a message from a client, then it expects to receive the message twice, once from the client and once from its buddy u'. If u receives a message from a parent node $p(u)$ or $p(u')$, then u expects to receive this message three times: from $p(u)$, $p(u')$, and u'. The message tag is deleted after u has received the messages from all the expected senders.

[1] Cross messages can be avoided if both nodes have performed their splits and have become internal nodes.

This mechanism lets u forward a message only once, although u receives that same message up to three times. However, unfortunately, it also introduces an extra complication in the following situation: Leaf u has performed a split, while u' has not. If u has sent the message to children v and v' while u' has sent an answer to the client c, then the following happens. By receiving the message from u, u' recognizes that its buddy has links to their children while it does not. To remedy the situation, u' now sends the message $Search(c, k, ID)$ to v and v' and changes its tag entry "Sent to" from (c, u) to (v, v', u). After that u' will perform a *split*, see Section 3.3. Hence, the search along connected paths in this way correctly manages the tags, provided that no crash failures occur.

Table 1 presents the pseudo-code for a node that is not a root. In this case, let s be the sender of a search message m, where s can be a parent node or the buddy. With p, we denote the parent of u. The pseudo-code for a root node is quite similar. The difference to the previous case is that the sender can be a client c or the buddy u' and that tags can be deleted after the second copy of the search request has been received.

Table 1. The pseudo-code for a search operation.

```
ReceiveSearchRequest(client c, key k, identifier ID)
    if (u is an inner node)
        if (there is no tag for m)
            generate a tag
            forward the message to children v, v' and to u'
        else (comment: there is no tag for m)
            if (the message has been received from p, p' and u')
                delete the tag
    else (comment: u is a leaf)
        if (s = u' and u' is an inner node)
            if (there is no tag for m)
                generate a tag
                forward the message to children v, v' and to u'
            else (comment: there is a tag for m)
                forward the message to children v, v'
                delete the tag
        if (s = u' and u' is a leaf)
            if (there is no tag for m)
                generate a tag
                forward the message to client c and to u'
            else (comment: there is a tag for m)
                if (the message has been received from p, p' and u')
                    delete the tag
        if (s = p or s = p')
            if (there is no tag for m)
                generate a tag
                forward the message to u' and send an answer to c
            else (comment: there is a tag for m)
                if (the message has been received from p, p' and u')
                    delete the tag
```

3.2.3. *Insert*. To insert key k into the HAT, client c sends the message *Insert*(c, k, ID) to the roots r_l, r_r. In the same way as a search message, this insert message is forwarded through the HAT. Each internal node u that receives *Insert*(c, k, ID) forwards the message to the buddy u' and to children v and v' with $k \in I_v$. A tag for this message is created.

Eventually the message will reach a leaf. If k is in the responsible leaf u, i.e., $k \in K_u$, then u sends the message *NotInserted*(k, ID) to c; otherwise k is inserted into u, i.e., $K_u \mapsto K_u \cup \{k\}$, and u sends the message *Inserted*(k, ID) to c.

Insertions into a leaf u could lead to a *split* of u, see Section 3.3.

3.3. *Split*

A split of a leaf u is necessary if the set of stored keys K_u and therefore the workload becomes too big. During a split, the leaf u becomes an internal node with left child v_1 and right child v_2.

Because a node and its buddy must have the same responsibility interval, u and u' must have the same split value $\sigma_u = \sigma_{u'}$. Since the split should be adaptive, the split value has to be chosen according to the key sets K_u and $K_{u'}$, for instance as their median key (a merely space-dependent split value such as $l_u + \lfloor (r_u - l_u)/2 \rfloor$ is not sufficiently adaptive). However, since insertions can be made in u and u' in a different order, K_u and $K_{u'}$ can differ greatly in size. Therefore, no node alone can choose the split value, we need a consensus between u and u'. In [4] it was shown that such a consensus between two or more nodes is impossible in our model; there is no protocol that guarantees consensus in the presence of failures.

We arrive at a split value decision by circumventing the impossibility of consensus: The split value is chosen by a central entity, the *split manager*. The split manager is stored on the fail-safe server S_0, and hence every node knows its address.

As a suggestion for the split value σ_u, u computes the median $\tilde{\sigma}_u$ of its key set K_u. Then u sends the message *SplitRequest*$(u, u', \tilde{\sigma}_u)$ to the split manager. After having received the first split request from u or its buddy u', the split manager selects four new nodes v_1, v_2, v_1', v_2' (according to the scheme described in Section 3.1), decides the split value σ_u (picking either $\tilde{\sigma}_u$ or $\tilde{\sigma}_{u'}$), and sends the message *SplitGrant*$(u, u', v_1, v_2, v_1', v_2', \sigma_u)$ to u.

The split manager responds with *SplitGrant*$(u', u, v_1', v_2', v_1, v_2, \sigma_u)$ if it receives a split request from u'. This implies that the split manager keeps track of that split, using a *split tag*. If the split manager receives another split request from u' or u, it uses the split tag to reconstruct the *SplitGrant* message sent before and to reuse the previously chosen split value. As before, to save memory, the split tags should be deleted as soon as they have become useless.

Again, this requirement introduces an extra difficulty. Because a leaf can perform many splits, it is not enough that a tag is deleted after each of both corresponding nodes has performed a split. To see this, assume that a leaf u had performed a split before it had a crash failure. Its buddy u' has not performed a split. If u sends a recover request to u', then u is recovered as a leaf. Therefore, u can split again, and it can repeat this over and over.

We solve this problem by keeping track of the split event history in the split tags. A split tag $t(u, u')$ consists of four parts:

- *Nodes to split*: The addresses of the nodes u, u'.
- *Split value*: The split value σ_u that the split manager selected.
- *New nodes*: The addresses of the four children v_1, v_2, v_1', v_2'.
- *Split performed*: Two flags indicating whether the split manager has received the forwarded messages about performed splits from u, u'. The flags are initialized as "no".

After having received the split grant, u uses the split value σ_u to compute new intervals $I_1 := [l_u, \sigma_u], I_2 := [\sigma_u+1, r_u]$ and key sets $K_1 := K_u \cap I_1, K_2 := K_u \cap I_2$. Then u sends the message $\mathit{Initialize}(u, u', v_i, v_i', s(v_i), s(v_i'), I_i, K_i)$ to v_i and the message $\mathit{Initialize}(u', u, v_i', v_i, s(v_i'), s(v_i), I_i, K_i)$ to v_i'. Then u stores the addresses of v_1, v_2 as its children and the addresses of v_1', v_2' as its buddy's children and deletes its key set K_u. Furthermore, u sends the message $\mathit{SplitPerformed}(u)$ to its buddy u'. Then u is an internal node.

Eventually u' receives the message $\mathit{SplitPerformed}(u)$. If u' is already an internal node, then u' forwards the message $\mathit{FwSplitPerformed}(u)$ to the split manager. If u' is still a leaf, $\mathit{SplitPerformed}(u)$ forces u' to begin its own split. After becoming an internal node, u' sends the message $\mathit{FwSplitPerformed}(u)$ to the split manager.

The $\mathit{Initialize}(u, u', v_i, v_i', s(v_i), s(v_i'), I_i, K_i)$ message reaches the server $S(v_i)$ eventually. If node v_i has not been initialized already by a message from u', then $S(v_i)$ initializes v_i with responsibility interval $I_{v_i} := I_i$ and key set $K_{v_i} := K_i$. The addresses $u, u', v_i', s(v_i), s(v_i')$ are the addresses of the parents, the buddy, the sibling, and its buddy's sibling. The buddy address v_i' is stored in the secure memory of $S(v_i)$. If a second message $\mathit{Initialize}(u', u, v_i, v_i', s(v_i), s(v_i'), I_i, K_i')$ is received, then K_i' is inserted in K_{v_i}, i.e., $K_{v_i} := K_{v_i} \cup K_i'$.

If the split manager receives a message $\mathit{FwSplitPerformed}(u)$, then it sets the corresponding entry in the split tag $t(u, u')$ to "yes". If both entries are set to "yes", then $t(u, u')$ is deleted.

As a detail on the side that illustrates the intricacy of our mechanism, note that since every node v knows the address of its buddy v', it could happen that v sends a message to v', although the server $S(v')$ does not know of v' yet. However, the server $S(v')$ stores the message in a large enough queue for future service. After v' has been initialized, it works on the messages that have been received before by $S(v')$. Therefore, we can safely assume that every node has a buddy at all times.

3.4. *Recovery*

Let S be a server after a crash, whose state changes from crashed to operational. Then S starts the following *recovery protocol*. According to the model, S knows the address of the buddy of at least one of its nodes from the secure memory. Let u' be one of these addresses, indicating that there was a node u on S before the crash. Every such node u on the server now sends the message $\mathit{RecoverRequest}()$ to its buddy u'. If u' is an internal node, it answers with the message $\mathit{Recover}(I_u, v_1, v_2, v_1', v_2', \sigma_u, p(u), p(u'), s(u), s(u'))$, where I_u is the responsibility interval of u and u'; v_1, v_2, v_1', v_2' are the addresses of the children

and their buddies; σ_u is the split value; $p(u)$, $p(u')$ are the addresses of the parents; and $s(u), s(u')$ are the addresses of the siblings. If u' is a leaf, then the answer is $Recover(I_u, p(u), p(u'), s(u), s(u'), K_{u'})$, where $K_{u'}$ is the set of keys of u', and the rest is as before. After having received and processed this recover message, server S checks whether there is a child v_i, a parent $p(u)$, or a buddy's sibling $s(u')$ that should be on S and that has no address of its buddy in the secure memory. If so, S initializes this node and sends the recover request to its buddy.

After every node on the server has received and processed the recover message from its buddy, the recovery protocol is finished.

Observe that it is enough for a server in the recovery process to know one node's buddy, even if many nodes were stored on that server before the crash. To see this, let u, v be two nodes on the same server, i.e., $S(u) = S(v)$. If both nodes are in the same tree, without loss of generality $u, v \in T_l$, then these nodes are on a path from the root to a leaf, i.e., one node is the descendant of the other. If $u \in T_l$ and $v \in T_r$, then there are nodes $w_l \in T_l$ and $w_r \in T_r$ with the properties:

- u is a descendant of w_l.
- v is a descendant of w_r.
- w_l is the sibling of the buddy of w_r, i.e., $w_l = s(w_r')$.

Therefore, the recovery protocol can recover every node on a server, if the address of only one buddy is available and none of these buddies has crashed.

4. Properties

We now argue that the mechanism described in Section 3 indeed leads to the desired behavior. Let H be a HAT with subtrees T_l, T_r. In its history, H starts out as an empty data structure with two roots r_l, r_r. Then a finite sequence of insert and search operations $O = \{o_1, o_2, \ldots, o_n\}$ is performed on H, and there are no other operations on H. During this phase, servers are allowed to have crash failures. Eventually, H reaches a state in which no messages are sent or received any more; measured in global time, we call this moment τ_0. Let us now observe H from an external point of view. We look at the set $\mathcal{S}$ of involved servers and the set $F \subset \mathcal{S}$ of servers that have had crash failures before τ_0. For this situation, we get the following results.

Theorem 4.1 (Isomorphic Trees and Overhead). *If no server has had a crash failure, i.e., $F = \emptyset$, then at τ_0 the HAT H is in the following situation*:

1. *The trees T_l, T_r are isomorphic.*
2. *For every pair of nodes u, u', $I_u = I_{u'}$ holds.*
3. *For every pair of leaves u, u', $K_u = K_{u'}$ holds.*
4. *All tags have been deleted.*

Furthermore, the number of messages sent to perform O on H is less than seven times the number of messages in each one of the underlying trees T_l or T_r.

Proof. Every insert operation is performed on both trees. Therefore, if one of the two roots has performed a split, the other has done it too. Furthermore, the splits have been done with the same split value. Induction shows the first three properties.

Let u, u' be a pair of leaves. The split of one node forces the other to split, too. The nodes u, u' perform their splits and send the messages *SplitPerformed* to its buddy, and they forward these messages to the split manager. The split manager has created the split tag $t(u, u')$, and after having received these *SplitPerformed* messages, $t(u, u')$ is deleted.

Let c be a client that sends an insert or search message to a pair of nodes u, u'. Both nodes create message tags, but these tags will be deleted after having received the second message from the buddy. If both nodes are leaves, then they send messages only back to c and no other tags are created. If both nodes have performed splits, then u, u' forward the messages to nodes v, v'. Each of these two nodes will receive messages from u, u' and one from its buddy. Therefore, the tags in v, v' will be deleted. If u has performed a split while u' has not, then u sends the message to nodes v, v' and u' sends a message back to c. The cross message $(Search(c, k, ID), v, v')$ forces u' to send the message to v, v' too. Therefore, each of the nodes v, v' receives three messages and deletes the corresponding tag after that.

We prove the factor 7 by counting the messages that are sent to perform the operations O. A message can be sent (1) from a node to another node, or (2) from a node to the split manager, or (3) from the split manager to a node, or (4) from a leaf to a client. Let $\mathcal{M}_H(O)$ be the set of these messages, and let $M_H(O) := |\mathcal{M}_H(O)|$. This number will be compared with the number of messages in the tree T_l. A message $m \in \mathcal{M}_H(O)$ is called an *internal message* of T_l if m is sent (1) from a node $u \in T_l$ to a node $v \in T_l$, or (2) from a node $u \in T_l$ to the split manager, or (3) from the split manager to a node $u \in T_l$, or (4) from a leaf of T_l to a client. To perform O on T_l, these internal messages have to be sent. In order to compare $M_H(O)$, let $M_l(O)$ be the number of internal messages of T_l. Lemma 4.2 shows that if no crash failure occurs, then $M_H(O) < 7M_l(O)$ holds. This completes the proof of the theorem. □

Lemma 4.2. *If no crash failure occurs, then* $M_H(O) < 7M_l(O)$ *holds.*

Proof. We regard every possible case and compute the factor that bounds the number of messages in this case.

Case 1. If u, u' are internal nodes that receive a message, e.g., a $Search(c, k, ID)$ message, for the first time, then due to their routing scheme u and u' send six messages. One of them is an internal message of T_l.

Case 2. If two leaves u, u' receive a message, then each sends two messages, one to the client c and one to the buddy. One of these four messages is counted in $M_l(O)$.

Case 3. It is possible that a node u has performed its split while u' has not. In this case at most seven messages are sent. Node u sends messages to two nodes v, v' and to u'; u' sends messages to u and to the client c; by receiving the message from u, u' recognizes that it has to send the message also to v and v'. One of these seven messages is an internal message. The same result holds for the case that u' has performed a split while u has not.

Case 4. If two leaves u, u' perform their splits 12 messages have to be sent. For each leaf there are the *SplitRequest*, *SplitGrant*, *SplitPerformed*, *FwSplitPerformed*, and two Initialize messages. Only 3 of these 12 messages are internal messages of T_l.

Case 3 leads to the inequality $M_H(O) \leq 7M_l(O)$, but, since for every operation $o \in O$ case 2 occurs once, the inequality becomes $M_H(O) < 7M_l(O)$. □

It is obvious that H works well if no crash failure occurs: every insertion and every search will be carried through, the trees tend to be isomorphic and every tag will be deleted. Deletion of the tags cannot be guaranteed, if failures can occur. However, it has to be guaranteed that a split tag is not deleted too early. Otherwise it could happen that the unique split value gets lost in a sequence of splits, crash failures, and recoveries, i.e., u uses a split value different from that of u'. This would imply that the buddy condition is violated, and recovery is not possible.

Lemma 4.3. *Let u, u' be two nodes, where one of them has sent a split request to the split manager. After the corresponding split tag $t(u, u')$ is deleted, neither u nor u' will send a split request.*

Proof. Assume that the tag $t(u, u')$ is deleted. Each node must have forwarded the split performed message from its buddy. At the moment at which u has sent the message *FwSplitPerformed*(u'), it has been an internal node already. If u' sends a recover request to u, then u' will be recovered as an internal node. Therefore, no node will send a split request. □

Definition 4.4 (Recoverable Crashes). We call the set of crashed servers F *recoverable*, if for every node u, $S(u) \notin F$ or $S(u') \notin F$ hold.

Note that in this very strict definition, we do not pay attention to the possibility that a crashed server could have performed its recovery before a second server crashes, without any loss of data or messages. However, on the other hand, in contrast to what one might think at first glance, the restriction that two buddies u, u' are not crashed at the same time, is not strict enough. This is shown in the example in Figure 7. Messages are sent to a node u and its buddy u'. The messages get lost although the nodes u, u' are not crashed at the same time, because the messages reach crashed servers $S(u)$, $S(u')$.

We only state the properties of the HAT under extreme circumstances; situations in between will lead to behavior in between.

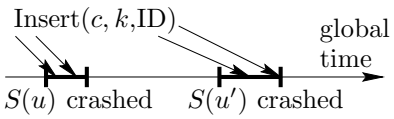

Fig. 7. Unrecoverable situation.

Theorem 4.5 (Correct Operation). *If F is recoverable, then all of the following hold at τ_0:*

1. *Every search request has been answered.*
2. *Every insert request has been processed by at least one responsible leaf.*
3. *Let k be a key that has been inserted. If no more crash failure happens after τ_0 and a single client c starts a search for k, then c will be informed that k has been found.*

Proof. We can assume that there is no recovery, i.e., crashed servers remain crashed. Let u, u' be the addressees of a $Search(c, k, ID)$ message. Since F is recoverable at least one of these nodes is operational and acts on the message: if it is a leaf, it sends a message to c, if it is an internal node it forwards $Search(c, k, ID)$ to the children v, v'. Since O is finite, the number of nodes in H is bounded and the message will reach a leaf eventually. This leaf sends a message back to the client c. The same holds for an $Insert(c, k, ID)$ message.

For every inserted key k there is a node u, with k inserted in u and $S(u) \notin F$. The $Search(c, k, ID)$ message will reach u eventually. If u is still a leaf, then u sends a response to c. If u has performed a split, then u forwards $Search(c, k, ID)$ to a child v. Eventually $Search(c, k, ID)$ reaches a leaf w. Since c has sent the $Search(c, k, ID)$ after τ_0, the key k is in w, i.e., $k \in K_w$. Then w sends a response to c. □

Theorem 4.6 (Trees are Identical Copies). *If F is recoverable and no more crash failure happens after τ_0, then H will eventually reach a state in which all of the following hold:*

1. *The trees T_l, T_r are isomorphic.*
2. *Both nodes in a pair of nodes u, u' have the same responsibility interval, i.e., $I_u = I_{u'}$.*
3. *Both nodes in a pair of leaves u, u' have the same data, i.e., $K_u = K_{u'}$.*

Proof. Let S be a server with $S \in F$. Due to the assumption that a server can have a crash failure only after the initialization of the first node, S had stored at least one node before the crash failure happened. Let u be such a node. Since F is recoverable, the server $S(u')$ had no crash failure. Therefore, u can determine its parent, children, and sibling. This is enough information for S to reconstruct every node v with $S = S(v)$. Therefore, the trees become isomorphic.

Let u, u' be two nodes. If $S(u) \notin F$ and $S(u') \notin F$, then $I_u = I_{u'}$ due to construction. If one of these nodes is on a crashed server, e.g., $S(u) \in F$, then this node is recovered with the responsibility interval of its buddy. Therefore, the equality $I_u = I_{u'}$ holds in both cases.

Let w be a node with $S(w) \notin F$. If $k \in K_w$, every request $Insert(c, k, ID)$ has reached w or w has been initialized with $k \in K_w$. A leaf splits when its key set becomes too big. Therefore, if w is a leaf, its decision to split or not to split does not depend on the size of F. This shows that $K_u = K_{u'}$ for a pair of leaves with $S(u) \notin F$ and $S(u') \notin F$. If $S(u) \in F$ and $S(u') \notin F$, then the key sets are equal, since K_u is a copy of $K_{u'}$. □

Let us summarize: a HAT works well, if not too many servers are crashed at the same time and recovery is performed fast enough. Of course, if crashes are too frequent and recovery is too slow, no data structure whatsoever can offer a reliable service.

5. Improvements and Modifications

So far we have described the HAT with a few assumptions and in its most simple form. We now discuss the flexibility of the basic HAT concept.

5.1. *Secure Memory and Early Crashes*

We have assumed that every server has a small secure memory. This is done to enable a server's recovery after a crash. However, the server can obtain the information in a different way, too. Let S be a server that after having a crash failure wants to recover. If S receives a message from a node u, then S knows that there should be a node $v \in S$ which is in some relation to u. Therefore S can send a message to u, asking for the information. Hence, an extra message exchange can replace the secure memory. The same technique can be applied to eliminate the assumption that no server crashes before it is involved in the data structure: if it instead does crash first, but later gets a message from some other server, it will know that it is involved, and it will get the necessary information with a message exchange.

5.2. *Lazy Update*

We described in Section 2 how clients with more activity tend to have better knowledge of the tree. We now explain the (fairly obvious) extension of the *lazy update concept* presented in [6] to the HAT. If a node u sends a message m to another node or to a client, u attaches its address and its responsibility interval to m. Every search or insert operation a client c initializes, gives information about the HAT's structure to c. If c knows the addresses of two nodes u, u' and their responsibility interval I_u, c can send every search for a key $k \in I_u$ to u, u' directly.

5.3. *Hidden Data*

In a distributed data structure, it is not always clear what the correct answer for a search request is. If one client searches for a key k and a second client wants to insert k into the structure, the answer to the search depends on factors such as the transmission time of the messages. This is a somewhat undesirable, but unavoidable, property. Therefore in a HAT, a client can get two different answers for a search. From one positive answer already, the client knows that the key is present.

In the described model, data structures have another property. It is possible that a key k is not found, although k has been inserted into a leaf earlier. This can happen if a leaf u has performed a split (and deleted its key set K_u), while the child has not received the *Initialize* message from u, i.e., the data is hidden in an unreachable message. In the following we describe a protocol that avoids this undesired behavior.

If a node u performs a split and sends *Initialize* messages to children v_i, v_i', then u does not delete its key set $K_u = K_1 \cup K_2$. Every following message *Search*(c, k, ID) is forwarded according to the described routing scheme, but additionally u sends an answer to c. Also, every following message *Insert*(c, k, ID) is forwarded according to the routing scheme, and k is inserted in K_u. Node u is called *schizophrenic*, since it acts on a leaf and an internal node at the same time.

If a node v receives an Initialize message, then it sends a message *InitializationOK*() back to the sender. If node u receives the message *InitializationOK*() from a node v_i or v_i', then it deletes K_i, i.e., $K_u \mapsto K_u \backslash K_i$. If $K_u = \emptyset$, then u stops its schizophrenic behavior.

It can be easily seen that if a key k has been inserted in a leaf u, then all following search requests will find at least one copy of k.

6. Conclusions

We have proposed a distributed dictionary that tolerates arbitrary single server crashes. The distinctive feature of our model is that the crash of a server cannot be detected. This is in contrast to all other proposals of distributed fault-tolerant search structures presented thus far. It reflects the real situation in a global database more accurately, and is in general more suitable to complex overall conditions. This makes our solution fundamentally different from all previous ones, but also more complicated. We have presented in detail the algorithms for searching, insertion, and graceful recovery of crashed servers.

The HAT structure works in a weak environmental setting. In order to get a better understanding of the HAT structure, experimental studies will be carried out in the future. Designing an experimental setup in itself is a major effort, since we need to appropriately intertwine a wealth of parameters, ranging from patterns for data, for queries, to timing and distribution of crash failures. Especially, we are interested in the probability of loss of data and in the overhead during the time period in which operations are performed. Furthermore, we want to compare different allocation rules that map nodes to servers.

We are working on the application of our scheme on a pair of B-trees (see [2]). We aim at a structure that is balanced and therefore guarantees logarithmic tree height. On the other hand, we expect disadvantages also: For instance, it may be impossible to delete message tags, because there can be more than one path from a root to a leaf. Furthermore, there is no guarantee that the path length is logarithmic although the tree height is. These issues have to be taken into consideration when generalizing our scheme to B-trees.

References

[1] Y. Breitbart, S. Das, N. Santoro, P. Widmayer (eds): *Distributed Data and Structures* 2, *DIMACS Workshop*, *Princeton*, 1999. Proceedings in Informatics 6. Carleton Scientific, Waterloo, Ontario, 1999.

[2] Y. Breitbart, R. Vingralek: Distributed and scalable B^+-tree data structures. In: [13], pp. 98–117.

[3] C.S. Ellis: Distributed data structures: a case study. *IEEE Trans. Comput.*, C-34: 1178–1185, 1985.

[4] M. Fischer, N. Lynch, M. Paterson: Impossibility of distributed consensus with one faulty process. *J. Assoc. Comput. Mach.*, 32(2): 374–382, 1985.
[5] J. Gray, A. Reuter: *Transaction Processing*: *Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, 1993.
[6] B. Kröll, P. Widmayer: Distributing a search tree among a growing number of processors. *Proc. ACM SIGMOD Conf. Management of Data*, pp. 265–276, 1994.
[7] B.W. Lampson: How to build a highly available system using consensus. *Proc. Workshop on Distributed Algorithms* (*WDAG*), pp. 1–17, 1996.
[8] W. Litwin, J. Menon, T. Risch, Th. Schwarz: Design issues for scalable availability LH* schemes with record grouping. In: [1], pp. 38–55.
[9] W. Litwin, M.-A. Neimat: High-availability LH* schemes with mirroring. *Proc. First Internat. Conf. on Cooperating Information Systems* (*CoopIS*), pp. 196–205, 1996.
[10] W. Litwin, M.-A. Neimat, D.A. Schneider: LH*—A scalable, distributed data structure. *ACM Trans. Database Systems*, 21(4): 480–525, 1996. A preliminary version appeared as: LH*—Linear hashing for distributed files. *Proc. ACM SIGMOD Conf. Management of Data*, pp. 327–336, 1993.
[11] W. Litwin, Th. Schwarz: LH^*_{RS}: A high-availability scalable distributed data structure using Reed Solomon codes. *Proc. ACM SIGMOD Conf. Management of Data*, 237–248, 2000.
[12] D. Lomet: Replicated indexes for distributed data. *Proc. Fourth Internat. Conf. Parallel and Distributed Information Systems* (*PDIS*), pp. 108–119, 1996.
[13] N. Santoro, P. Widmayer (eds): *Distributed Data and Structures* 1, *IPPS Workshop*, *Orlando*, *Florida*, 1998. Proceedings in Informatics 2. Carleton Scientific, Waterloo, Ontario, 1999.
[14] K. Schlude, E. Soisalon-Soininen, P. Widmayer: Distributed highly available search trees. *Proc. SIROCCO* 9, pp. 259–274, 2002.
[15] R. Vingralek, Y. Breitbart, G. Weikum: Snowball: Scalable storage on networks of workstations with balanced load. *Distrib. Parallel Databases* 6: 117–156, 1998.

Received September 12, 2002 *and in final form May* 15, 2003. *Online publication October* 6, 2003.